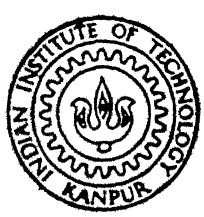
A SYSTEM FOR OBJECT AND RULE ORIENTED PROGRAMMING

by

P. DURGA PRASADA RAO

Th
CSE/1988/M
R 18 S
CSE
988
M
RAO
R8



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY 1988

A SYSTEM FOR OBJECT AND RULE ORIENTED PROGRAMMING

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

P. DURGA PRASADA RAO

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1988

13 APR 1989
CENTRAL LIBRARY
I. I. T. KANPUR
Acc. No. A104143

CERTIFICATE

This is to certify that the work entitled **A SYSTEM FOR OBJECT AND RULE ORIENTED PROGRAMMING** by P.Durga Prasada Rao has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

T.V.Prabhakar,
Asistant Professor,
Dept. of C.S.&Engg,
I.I.T.,Kanpur.

Date: Feb 1988.

ACKNOWLEDGEMENTS

I am deeply indebted to Dr. T.V.Prabhakar for his able guidance, cooperation and constant encouragement in making this thesis possible. He has been very understanding and his approach has been always humane.

I thank Dr.H.Karnick for providing me with literature on 'LOOPS'.

I thank Dr.Rajiv Sangal for introducing me to the field of Artificial Intelligence.

I thank Bobrow and Stefik. I borrowed the concept of combining object oriented programming and rule oriented programming from their work.

I thank Khadilkar for his help regarding my thesis.

My thanks are also due to all my friends who have helped to make my stay at IIT Kanpur pleasant and memorable.

(P.D.Prasada Rao)

CONTENTS

1. Introduction 1

1.1 The Rationale for an Intergrated Programming Environment ... 3

1.2 Overview of the Report 4

2. Object-Oriented Programming 5

2.1 Structure of Classes and Instances 5

2.1.1 Classes 5

2.1.2 Variables 6

2.1.3 Methods 6

2.1.4 Metaclasses 7

2.1.5 An Example of a Class 7

2.2 Inheriting Variables and Methods 8

3. Rule-Oriented Programming 11

3.1 Organizing a Rule-Oriented Program 13

3.2 Control Structures for Selecting Rules 15

3.3 One-Shot Rules 18

3.4 Comparison with Other Rule Languages 19

3.4.1 The Rationale for RuleSet Hierarchy 19

3.4.2 The Rationale for RuleSet Control Structures ... 20

4. The Language 22

4.1 The Object Oriented Programming Constructs 22

4.1.1 Send 22

4.1.2 The Access Functions 28

4.2 The Rule Language 33

4.2.1 Form of a Rule 33

4.2.2 Kinds of Variables 35

4.2.3 Creating a RuleSet 36

4.2.4 Invoking a RuleSet 37

4.2.5 Editing a RuleSet 38

4.2.5.1 Editing a Rule 38

4.2.5.2 Addition / Deletion of a Rule 40

4.2.6 Viewing a RuleSet 41

4.2.7 The 'Stop' Action....42
4.3 Saving a Session....42
5 Examples....43
5.1 Example-1....43
5.2 Example-2....51
5.3 Example-3....54
6. The Implementation....63
6.1 Data Structures....63
6.1.1 Metaclasses....63
6.1.2 Classes....64
6.1.3 Instances....66
6.1.4 RuleSets....67
6.1.5 Global Variables....68
6.2 OBJECT, CLASS, METACLASS and RULESET....70
6.2.1 Methods of OBJECT....73
6.2.2 Methods of METACLASS....73
6.2.3 Methods of CLASS....73
6.2.4 Methods of RULESET....75
6.3 Send - Invoke Algorithm....75
6.4 Loopsout Algorithm....77
6.5 How to Start the System....78
6.6 A Note....78
7. Conclusions and Suggestions for Further Work....80
References....83
Summary of Commands....84

Chapter 1

INTRODUCTION

Four distinct paradigms of programming available in the Computer Science community today are oriented around procedures, objects, data access and rules. Usually these paradigms are embedded in different languages. Our system is designed to incorporate procedures, objects and rules within the Lisp programming environment, to allow users to choose the style of programming which best suits their application.

PROCEDURE ORIENTED PROGRAMMING

Lisp is a procedure oriented language; the procedure oriented paradigm is the dominant one provided in most programming languages today. Two separate kinds of entities are distinguished: procedures and data. Procedures are active and data are passive. The ability to compose procedures out of instructions and to invoke them is central to organizing programs using these languages. Side effects happened when separate procedures share a data structure and change parts of it independently.

OBJECT ORIENTED PROGRAMMING

This paradigm was pioneered by Smalltalk, and has its

roots in SIMULA and in the concept of data abstraction. In contrast with the procedure-oriented paradigm, programs are not primarily partitioned into procedures and separate data. Rather, a program is organized around entities called objects that have aspects of both procedures and data. Objects have local procedures (methods) and local data (variables). All of the action in these languages comes from sending messages between objects. Objects provide local interpretation of the message form.

The object-oriented paradigm is well suited to applications where the description of entities is simplified by the use of uniform protocols. For example in a graphics application, windows, lines and composite structures could be represented as objects that respond to a uniform set of messages(i.e., Display, Move, and Erase). An important feature of these languages is an inheritance network, which makes it convenient to define objects which are almost like other objects. This works together with the use of uniform protocols because specialized objects usually share the protocols of their super classes.

RULE ORIENTED PROGRAMMING

In rule oriented programming, the behavior of the system is determined by sets of condition-action pairs. These *RuleSets* play the same role as subroutines in the procedure oriented metaphor. Within a RuleSet, invocation of rules is guided largely by patterns in the data. In the typical case, rules correspond to nearly-independent patterns in the data.

The rule-oriented approach is convenient for describing flexible responses to a wide range of events characterized by the structure of the data.

DATA ORIENTED PROGRAMMING

In both of the previous paradigms, the invocation of procedures (either by direct procedure call or by message sending) is convenient for creating a description of a single process. In data oriented programming, action is potentially triggered when data are accessed.

Programs are easier to build in a language when there is an available paradigm that matches the structure of the problem. A variety of programming paradigms gives breadth to a programming language. The paradigms described here offer distinct ways of partitioning the organization of a program, as well as distinct ways of viewing the significance of side effects.

The intellectual precursors for this thesis are two languages - Smalltalk and Loops. Smalltalk is an object-oriented language. Loops is a language that combines the paradigms of procedure-oriented programming, object-oriented programming, data-oriented programming and rule-oriented programming.

1.1 THE RATIONALE FOR AN INTEGRATED PROGRAMMING ENVIRONMENT

RuleSets in our language are integrated with procedure-oriented, and object-oriented programming para-

digms. In contrast to single-paradigm rule systems, this integration has two major benefits. It facilitates the construction of programs which don't entirely fit the rule-oriented paradigm. Rule-oriented programming can be used selectively for representing just the appropriate decision-making knowledge in a large program. Integration also makes it convenient to use the other paradigms to help organize the interactions between RuleSets.

Using the object-oriented paradigm, RuleSets can be invoked as methods for the objects. The use of object-oriented paradigm is facilitated by special RuleSet syntax for sending messages to objects, and for manipulating the data in the objects.

1.2 OVERVIEW OF THE REPORT

Chapter 2 describes the object-oriented programming paradigm.

Rule-oriented programming paradigm is explained in the Chapter 3 of this report.

Chapter 4 describes the language that was implemented and the various constructs that are available to the user.

Examples are shown in the Chapter 5.

The various details of the implementation of the language are given in the Chapter 6.

Chapter 7 concludes the report and gives suggestions for future work.

Chapter 2

OBJECT-ORIENTED PROGRAMMING

Of late, there has been a lot of interest in the concept of object-oriented programming. New object-oriented languages and commercial environments continue to appear and a few of them are now becoming available for personal computers.

Object-based systems have had an anomalous history. They have their roots in Smalltalk-80, the Xerox-supported environment. Smalltalk-80 was developed and promoted as an internal vehicle for research at Xerox PARC and was only disclosed as a complete environment after ten years of development effort. During this time, no attempt was made to stimulate external discussion on its basic concepts. Other object-oriented languages, such as Simula, benefited from a more open research process.

2.1 STRUCTURE OF CLASSES AND INSTANCES

2.1.1 CLASSES

A *class* is a description of one or more similar objects. An *instance* is an object described by a particular class. Every *instance* has a name. *Classes* themselves are instances of a metaclass, usually the one called

CLASS. A *Metaclass* is a class whose instances are classes.

All the instances of a class have the same message interface; the class describes how to carry out each of the operations available through that interface. Each operation is described by a method. The selection of a message determines what type of operation the receiver should perform, so a class has one method for each selector in its interface.

2.1.2 VARIABLES

There are two kinds of variables - *Class variables* and *Instance variables*. *Class variables* are used to contain information about a class taken as a whole. *Instance variables* contain the information specific to an instance. Both kinds of variables have names and values. A class describes the structure of its instances by specifying the names and default values of instance variables. For example, the class *Point* might specify two instance variables, *x* and *y* with default values of Ø, and a class variable, *lastselectedpoint*, used by methods associated with all instances of class *Point*.

2.1.3 METHODS

A class specifies the behaviour of its instances in terms of their response to *messages*. The class associates *selectors* which are Lisp atoms, with *methods*, the Lisp functions that respond to the messages. All instances of a class use the same selectors and methods. Any difference in

response by two instances of the same class is determined by a difference in the values of their instance variables. For example, *Printon* can be used as a selector for the message which knows how to print out a file on a printer.

2.1.4 METACLASSES

Classes themselves are instances of some class. When we want to distinguish classes whose instances are classes, we call them metaclasses. When a class is sent a message , its metaclass determines the response if there is no method corresponding to that message in the class and supers of the classes. For example, instances of a class are created by sending the class the message *new*. For most classes, this method is provided by the standard metaclass for classes : CLASS. The user can create other metaclasses to perform specialized initialization.

2.1.5 AN EXAMPLE OF A CLASS

Let us look at the details of a class *areabudget*. This class inherits variables and methods from both of its super classes *ownedobject* and *budget*. In this example, the new class variable *maxbase* is introduced, and six instance variables *owner*, *base*, *overhead*, *employees*, *manager*, and *total* are defined. The Methods declaration names the Lisp functions that implement the methods. For example, *areabudget~report* is the name of a function that implements the *report* method for instances of *areabudget*.

Class - areabudget

Metaclass - CLASS

Supers - (ownedobject budget)

Class Variables - ((maxbase 25000))

Instance Variables -

```
( owner
  (base 1000)
  (overhead 2.25)
  employees
  manager
  total )
```

Methods -

```
((report areabudget~report)
 (storebase areabudget~storebase))
```

2.2 INHERITING VARIABLES AND METHODS

Inheritance is an important tool for organizing information in objects. It enables the easy creation of objects that are *almost like* other objects with a few incremental changes. Inheritance avoids the user to specify redundant information and simplifies updating, since information that is common need be changed in only one place.

The objects in the system exist in an *inheritance network* of classes. An object inherits its instance variable description and message responses. All descriptions in a class are inherited by a subclass unless overridden in the subclass. Each class can specify inheritance of structure and behaviour from any number of super classes in its supers list.

HIERARCHY

In the simplest case, each class specifies only one super class. If the class A has the supers list (B), a one element list containing B, then all of the instance variables specified local to A are added to those specified for B, recursively. That is, A gets all those instance variables described in B and all of B's supers. In this case one obtains strict inheritance hierarchy .

Any conflict of variable names is resolved by using the description closer to A in traversing up the hierarchy to its root at the class OBJECT. Looking up for a mehtod in the hierarchy also uses the same conflict resolution. The method to respond to a message is obtained by first searching A, then B, and then searching recursively in B's supers list.

AN EXAMPLE

Class	Super	Instance Variables	Methods
OBJECT	NIL	none	(s4 m6)
C	OBJECT	(w 7)	(s2 m4) (s3 m5)
B	C	(y 4) (z 3)	(s1 m2) (s2 m3)
A	B	(x 1) (y Ø)	(s1 m1)

In the definitions given in the above chart, an instance of A would be given four instance variables, w, y, z, and x in that order. The default value for y would be Ø, which overrides the default value of y inherited from B. The

instance would also respond to the four messages with selector s1, s2, s3, and s4. The method used for responding to s1 is m1, which is said to override m2 as the implementation of the message s1. Similarly, m3 overrides m4 as the implementation of message s2. The root class in the system, OBJECT, has no super class. All classes in the system are subclasses of OBJECT, directly or indirectly.

MULTIPLE SUPER CLASSES

In the system, the classes can have more than one class specified on their supers list. Multiple super classes admit a modular programming style where (i) methods and associated variables for implementing a particular feature are placed in a single class and (ii) objects requiring combinations of independent features inherit them from multiple supers. If D had the supers list (E A), first the description from E and its supers would be inherited, and then the description from A and its supers. In the simplest usage, the different features have unique variable names and selectors in each super. In case of a name conflict, the system uses a depth-first left to right precedence. For example, if any super of E had a method for s1, then it would be used instead of the method m1 form A. In every case, inheritance from OBJECT (or any other common super class) is only considered after all other classes on the recursively defined supers list.

Chapter 3

RULE-ORIENTED PROGRAMMING

The core of decision-making expertise in many kinds of problem solving can be expressed succinctly in terms of rules. The following sections describe facilities in the system for representing rules, and for organizing knowledge-based systems with rule-oriented programming. The rule language and programming environment are integrated with the object-oriented, and procedure-oriented parts of the system.

Rules in the system are organized into production systems (called RuleSets) with specified control structures for selecting and executing the rules. The work space for RuleSets is an arbitrary object.

Production rules have been used in expert systems to represent decision-making knowledge for many kinds of problem-solving. Such rules (also called *if-then* rules) specify actions to be taken when certain conditions are satisfied. Several rule languages (e.g., OPS5, ROSIE, AGE) have been developed in the past few years and used for building expert systems.

The system has the following major features for rule-

oriented programming :

(1) Rules are organized into ordered sets of rules (called RuleSets) with specified control structures for selecting and executing the rules. Like subroutines, RuleSets are building blocks for organizing programs hierarchically.

(2) The work space for rules in the system is an arbitrary object that was defined. The names of the instance variables provide a name space for variables in the rules.

(3) Rule-oriented programming is integrated with object-oriented and procedure-oriented programming in the system.

(4) RuleSets can be invoked in several ways : In the object-oriented paradigm, they can be invoked as methods by sending messages to objects. They can also be invoked directly from Lisp programs. This integration makes it convenient to use the other paradigms to organize the interactions between Rulesets.

(5) RuleSets can also be invoked from rules either as predicates on the LHS of rules, or as actions on the RHS of rules. This provides a way for RuleSets to control the execution of other RuleSets.

(6) The rule language provides a concise syntax for the most common operations.

Rules express the conditional execution of actions.

They are important in programming because they can capture the core of decision-making for many kinds of problem-solving.

3.1 ORGANIZING A RULE-ORIENTED PROGRAM

In any programming paradigm, it is important to have an organizational scheme for composing large systems from smaller ones. It is important to have a method for partitioning large programs into nearly-independent and manageably-sized pieces. In the procedure-oriented paradigm, programs are decomposed into procedures. In the object-oriented paradigm, programs are decomposed into objects. In the rule-oriented paradigm, programs are decomposed into *RuleSets*.

There are two approaches to organizing the invocation of RuleSets in the system :

Procedure-oriented Approach:

This approach is analogous to the use of subroutines in procedure-oriented programming. Programs are decomposed into RuleSets that call each other and return values when they are finished. SubRuleSets can be invoked from multiple places. They are used to simplify the expression in rules of complex predicates, and actions.

Object-oriented Approach:

In this approach, RuleSets are installed as methods for objects. The value computed by the RuleSet is returned

as the value of the message sending operation.

These approaches for organizing RuleSets can be combined to control the interactions between bodies of decision-making knowledge expressed in rules.

AN EXAMPLE

Let us look at a RuleSet of consumer instructions for testing a washing machine. The work space for the RuleSet is an instance of the class *washingmachine*. The control structure *while1* loops through the rules trying an escalating sequence of actions, starting again at the beginning if some rule is applied. Some rules, called one-shot rules, are executed at most once. These rules are indicated by the preceding '1'.

Rule 1 says that if the washing machine is operational, then stop (i.e. there is no need for executing the other rules of RuleSet). Rule 2 says that if the load of washing machine (which is stored in the variable name *load* of washing machine) is more than 1, then reduce the load to rectify the defect in the washing machine. Rule 3 says that if the washing machine is not plugged in, then plug in. Rule 4 says that if the voltage is zero, then reset the machine. Rule 5 says that if the input voltage is more than 11Ø, then call *pge*. Rule 6 says that if all the above rules fail, then call the dealer requesting service for the washing machine. Rule 7 says that if rules 1 to 6 fail, send a complaint to the manufacturer. Rule 8 says that if no

response from the manufacturer, then complain to the consumer board. Rule 9 says that there is nothing else the consumer can do regarding his/her washing machine and stop.

The details of the constructs available for RuleSets are explained in the Section 4.2 of Chapter 4.

```
RuleSet Name : checkwashingmachine;
Control Structure : while1;
While Condition : (not (get 'checkwashingmachine '*ruleapplied
(* What a consumer should do when a
                washing machine fails. *)
1. (((send self 'operational))
          ((stop)))
2. (((> (getvalue self 'load) 1))
          ((send self 'reduceload)))
3. (((not (getvalue self 'pluggedinto)))
          ((send self 'plugin)))
4. (1 ((eq Ø (getvalue (getvalue self 'pluggedinto) 'voltage)))
          ((send (getvalue self 'breaker) 'reset)))
5. (1 ((> 11Ø (getvalue (getvalue self 'pluggedinto) 'voltage)))
          ((send 'pge 'call)))
6. (1 (t)
          ((send (getvalue self 'dealer) 'requestservice)))
7. (1 (t)
          ((send (getvalue self 'manufacture) 'complain)))
8. (1 (t)
          ((send (getvalue self 'consumerboard) 'complain)))
9. (1 (t)
          ((stop)))
```

3.2 CONTROL STRUCTURES FOR SELECTING RULES

RuleSets in the system consist of an ordered list of rules and a control structure. Together with the contents of

the rules and the data, a RuleSet control structure determines which rules are executed. Execution is determined by the contents of rules in that the conditions of a rule must be satisfied for it to be executed. Execution is also controlled by data in that different values in the data allow different rules to be satisfied. Criteria for iteration and rule selection are specified by a RuleSet control structure. There are four primitive control structures for RuleSets in the system which operate as follows :

do1 :

The first rule in the RuleSet whose conditions are satisfied is executed. The value of the RuleSet is the value of the rule. If no rule is executed, the RuleSet returns NIL. The *do1* control structure is useful for specifying a set of mutually exclusive actions, since at most one rule in the RuleSet will be executed for a given invocation. When a RuleSet contains rules for specific and general situations, the specific rules should be placed before the general rules.

doall :

Starting at the beginning of the RuleSet, every rule is executed whose conditions are satisfied. The value of the RuleSet is the value of the last rule executed. If no rule is executed, the RuleSet returns NIL. The *doall* control structure is useful when a variable number of additive actions are to be carried out,

depending on which conditions are satisfied. In a single invocation of the RuleSet, all of the applicable rules are fired.

while1 :

This is a cyclic version of *do1*. If the while-condition is satisfied, the first rule is executed whose conditions are satisfied. This is repeated as long as the while condition is satisfied or until a *stop* statement is executed. The value of the RuleSet is the value of the last rule that was executed, or NIL if no rule was executed.

whileall :

This is a cyclic version of *doall*. If the while-condition is satisfied, every rule is executed whose conditions are satisfied. This is repeated as long as the while-condition is satisfied or until a *stop* statement is executed. The value of the RuleSet is the value of the last rule that was executed, or NIL if no rule was executed.

The *while-condition* is specified in terms of the variables and constants accessible from the RuleSet. Every time, the *while-condition* is evaluated and checked if it is getting satisfied or not. So, the *while-condition* should be an evaluable s-expression. The constant t can be used to specify a RuleSet that iterates forever (or until a *stop* statement is executed). Every RuleSet has a property

**ruleapplied* whose value is *t* if some rule was executed in the last iteration else NIL. The example RuleSet given in the section 3.1 uses this property **ruleapplied* in the *while-condition*.

3.3 ONE-SHOT RULES

One of the design objectives of the language is to clarify the rules by factoring out control information whenever possible. This objective is met in part by the declaration of a control structure for RuleSets.

Another important case arises in cyclic control structures which some of the rules should be executed only once. This is required to prevent the RuleSet from going into an infinite loop of executing the same rule in every iteration. Such rules are also useful for initializing data for RuleSets.

In the absence of special syntax, it would be possible to encode the information that a rule is to be executed only once as follows :

```
Control Structure :  while1
Temporary Vars :  triedrule3;
...
(((not triedrule3) condition1 condition2)
                ((setq triedrule3 t) action1))
```

In this example, the variable *triedrule3* is used to control the rule so that it will be executed at most once in an invocation of a RuleSet. However, the prolific use of rules with such control clauses in large systems has led to the common complaint that control clauses in rule languages

defeat the expressiveness and conciseness of the rules. For the case above, our language provides a shorthand notation as follows :

(1 (condition1 condition2) (action1))

The '1' in the beginning indicates that the rule should be executed only once in an invocation of a RuleSet. These rules are called *one-shot* rules.

3.4 COMPARISON WITH OTHER RULE LANGUAGES

This section considers the rationale behind the design of the rule language, focusing on ways that it diverges from other rule languages.

3.4.1 THE RATIONALE FOR RULESET HIERARCHY

The major features of a traditional production system are its flatness and having no organization in the rules. A traditional production system is a single set of rules. The flat organization is sometimes said to make it easy to add rules. But Bobrow and Stefik [3] observe that there is no inherent property of production systems that can make rules additive. Rather, *additivity* is a consequence of the independence of particular sets of rules. Such independence is seldom achieved in large sets of rules. When rules are dependent, rule invocation needs to be carefully ordered. Advocates of a flat organization tend to organize large programs as a single very large production system. In practice, most builders of production systems have found it essential to create groups of rules.

3.4.2 THE RATIONALE FOR RULESET CONTROL STRUCTURES

Production languages are sometimes described as having a *recognize-act cycle*, which specifies how rules are selected for execution. An important part of this cycle is the *conflict resolution strategy*, which specifies how to choose a production rule when several rules have conditions that are satisfied. For example, the OPS5 production language has a conflict resolution strategy (MEA) which prevents rules from being invoked more than once, prioritizes rules according to the recency of a change to the data, and gives preference to production rules with most specific conditions.

In our language, there are a small number of specialized control structures instead of a single complex conflict resolution strategy. These control structures are similar to the control structures of the familiar programming languages. For example, *do1* is like Lisp's COND, *doall* , is like Lisp's PROG, *whileall* is similar to WHILE statements in many programming languages.

The specialized control structures are intended for concisely representing programs with different control relationships among the rules. For example, the *doall* control structure is useful for rules whose effects are inteded to be additive and the *do1* control structure is appropriate for specifying mutually exclusive actions.

It could be argued that the *do1* and *doall* control

structures are not strictly necessary because such RuleSets can always be written in terms of *whilel* and *whileall*. Following this reductionism to its end, we can observe that every RuleSet could be re-written in terms of *whileall*. But, to achieve the concise expression in the rules, all four control structures are provided to the user.

Chapter 4

THE LANGUAGE

In this chapter, we present the language. There are two major parts in the language. They are *object oriented programming constructs* and *the rule language.*

4.1 THE OBJECT ORIENTED PROGRAMMING CONSTRUCTS

In our system, as almost everything is an object, the communication channel between the objects is vital to the system. The *send* command acts as the communication channel between the objects.

4.1.1 SEND

THE SYNTAX

(send <object> <selector> <arg1> ... <argn>)

Here, the message <selector> is passed to the object that is in the variable <object> with arguments as the parameters. All the arguments to the function *send* are evaluated.

THE COMMAND send AND THE STANDARD METHODS

(1) (send <metaclassname> 'new <classname> <superslist>)

This command creates a class with name equivalent to

<classname> under <metaclass>. The <metaclass> can be the system class *CLASS* or one of the user defined metaclasses. If <metaclass> is *METACLASS*, then this command creates a metaclass. <superslist> contains the names of classes that should be supers of the class defined. This command does not do anything about instance variables, class variables, methods etc. There are other standard ways of using *send* command to handle these things.

Example.-

<1> (send 'CLASS 'new 'student '(humans indian))

<2> (send 'METACLASS 'new 'animal ())

<3> (send 'animal 'new 'dog ())

Command <1> creates a class *student* under the system class CLASS with classes *humans* and *indian* as the supers of it. Command <2> creates a metaclass called *animal*. Command <3> creates a class *dog* under the metaclass *animal*.

(2)(send <classname> 'new <instancename>)

This command creates an instantiation of <classname> with name <instancename>. After this , one has to use the access function *putvalue* to give values to the instance variables.

Example.-

<-> (send 'dog 'new 'dogmatix)

This creates an instantiation with name *dogmatix* under the class *dog*.

(3) (send <classname> 'newwithvalues <instancename> <var-val list>)

<var-val list> looks like ((var1 val1) ...(varn valn)). This command creates an instance with name <instancename> under the class <classname> with its variables var1, .. ,varn initialized to val1, ..,valn.

(4) (send <classname> 'definstvars)

This command asks the user interactively to define instance variables for the class <classname>.

Example.- A sample session can be like the following -

```
<-> (send 'student 'definstvars) [ret]
    (Please give in the form of (variable initial val) or
                                <variable> one by one)
    (End it by typing END)
    Rollno [ret]
    (Institution IITK) [ret]
    END [ret]
<->
```

The session shown above defines two instance variables *Rollno* and *Institution* for the class *student*. The initial value for variable *Institution* is defined as *IITK*.

<5> (send <classname> 'defclvars)

This command asks the user interactively to define class variables for the class <classname>.

Example.-

```
<-> (send 'student 'defclvars) [ret]
    (Please give in the form of (variable initialval)
                                                one by one)
    (End it by typing END)
    (IQ high) [ret]
    (maturity low) [ret]
    END [ret]
<->
```

The session shown above definess two class variables *IQ* and *maturity* with values 'high' and 'low' respectively.

(6) (send <classname> 'defmethod <selector> <fnname>)

This command defines that the function <fnname> is to be executed whenever a message <selector> is received by the class <classname>. <fnname> can be an user defined function.

Example.-

```
<-> (send 'student 'defmethod 'gradecard 'pri-gr-cd) [ret]
```

Whenever a message *gradecard* is received by the class *student,* the function *pri-gr-cd* is executed with proper inputs.

(7) (send <classname> 'defmethod <selector> <listofargs> <form>)

This is another form of *defmethod.* Here, instead of giving the function name, the form (the form is a list of s-expressions) and the list of input arguments required for that list of s-expressions <listofargs> are given. If

<selector> is received by <classname>, then <form> is executed.

Example.-

<-> (send 'number-sys 'defmethod 'add '(a b)
'((print (+ a b)))) [ret]

(8) (send <classname> 'rmmethod <selector>)

This results in removing the method corresponding to <selector> from the class <classname>. If <classname> is the name of an instantiation, then the method for <selector> is removed from the class of that particular instance.

Example.-

<-> (send 'student 'rmmethod 'gradecard)

Sending the message *gradecard* to the class *student* after the execution of the above command results in an error.

(9) (send <classname> 'addsuper <superlist>)

<superlist> contains the list of class names that are to be added to the supers list of <classname>.

Example.-

<-> (send 'student 'addsuper '(disciple))

Even if the number of classes that are to be added as supers to <classname> is one, that single class should be put in a list.

(10) (send <classname> 'rmsuper <superlist>)

<superlist> contains the list of class names that are

to be deleted from the supers list of <classname>.

Example.-

<-> (send 'student 'rmsuper '(disciple))

(11) (send <classname> 'rmsub <sublist>)

<sublist> is the list of class names which are to be deleted from the sub-class list of <classname>. In turn, the classes whose names are in the <sublist> will no longer have <classname> as one of their supers.

Example.-

<-> (send 'disciple 'rmsub '(student))

(12) (send <classname> 'pclass)

This prints the details of the class <classname> on the terminal. The details include instance variables, class variables, super-classes, sub-classes and methods.

(13) (send <instance> 'pinstace)

This command is useful in seeing the details of an instance <instance>. This prints the class name to which <instance> belongs, and also the instance variables of <instance> along with their values.

(14) (send <class> 'listinst)

This command prints all the names of instances that were defined under the class <class>.

(15) (send <metaclass> 'classlist)

This command prints all the names of classes that were defined under the metaclass <metaclass>.

(16) (send <object> 'destroy)

This command results in totally removing the object <object> from the working environment. <object> can be an instance, a class, a metaclass or a RuleSet. If <object> is a class, all the instances defined under that class would also be destroyed. Similarly, if <object> is a metaclass, all the classes defined under that metaclass are also destroyed along with the metaclass.

4.1.2 THE ACCESS FUNCTIONS

In our system, the access functions are not implemented as standard system methods. Instead, they are implemented as function calls. The reason for this is the wide usage of the access functions in a session. Implementing the access functions as standard methods would have meant one extra pass through the inheritance structure for that object which we want to access or modify.(A pass through inheritance structure would be a depth-first search). Instead, the functions corresponding to the access functions take the required parameters and directly handle the necessary data structures. Only disadvantage with this implementation is that the uniformity is lost as these access calls are not passed as messages. But, the time we gain over loosing uniformity is significant.

(1) getvalue

(getvalue <object> <variable name>)

This command returns the value of the instance variable <variable name> in the object <object>. Each instance of a class has its own separate set of instance variables. If no local value of the variable <variable name> in the instance <object> is set, *getvalue* returns the default value from the class or from the supers of the class in case the immediate class does not have a default value for the variable <variable name>.

The function *getvalue* fetches a value from an instance of a class. It is an error to try to use *getvalue* to fetch an instance variable from a class. To fetch the default value of an instance variable from a class, *getclassiv* should be used.

Example :

```
(getvalue 'Vijay 'rollno)
```

This gives the value of the instance variable *rollno* of the instance *Vijay*.

(2) getclassvalue

(getclassvalue <object> <variable name>)

This command returns the value of the class variable <variable name> for the class of the object <object> (which may be either an instance or a class).

Class variables are inherited from the super classes. If <object> is an instance, lookup begins at the class of <object> since instances do not have class variables stored

locally. If the class does not have a class variable <variable name>, *getclassvalue* searches through the super classes of the class until it finds <variable name>.

Conceptually, one should think of a class variable of a class as being shared by all instances of that class, and by all instances of any of its subclasses. For example, suppose 'Transistor' is a class with class variable 'TransNum', and 'DepletionTransistor' is a subclass of 'Transistor'. Then setting the class variable 'TransNum' from an instance of 'DepletionTransistor' would be seen by all instances of 'Transistor'.

Example :

```
(getclassvalue 'human 'no-of-eyes)
```

This gives the value of class variable *no-of-eyes* of the class *human*

(3) putvalue

(putvalue <object> <variable name> <newvalue>)

This command stores <newvalue> as the value of the instance variable <variable name> in the object <object>. *putvalue* works for storing value in an instance of a class. It is an error to try to store a default instance variable in a class with *putvalue*. To store the default value for an instance variable directly in the class, *putclassvalue* should be used.

Example.-

```
(putvalue 'position 'x 0)
```

(4) putclassvalue

(putclassvalue <object> <variable name> <newvalue>)

It is similar to *putvalue*, except that it stores <newvalue> as the value of a class variable. <object> may either be an instance or a class. It returns the <newvalue>.

If <variable name> is not local to the class, then the value will be put in the first class in the inheritance list that <variable name> is found.

(5) getclassiv, putclassiv

(getclassiv <object> <variable name>)

(putclassiv <object> <variable name> <newvalue>)

These commands are for dealing with the default values of the instance variables of a class.

(6) getivhere, getcvhere

(getivhere <instance> <variable>)

(getcvhere <class> <variable>)

These two commands are for the *Local Get* operations. *getivhere* returns the instance variable value that is found in the instance. *getcvhere* returns the class variable value that is found in the class. In both of these cases, inheritance is not applied.

(7) getclass

(getclass <instance>)

This returns the name of the class to which <instance> belongs.

(8) getmetaclass

(getmetaclass <class>)

This command returns the name of the metaclass to which <class> belongs.

(9) getmethod

(getmethod <class> <selector>)

This command returns the method (Franz-Lisp function name) which implements the message <selector> of the class <class>. One can see the definition of the function by using the Lisp command 'pp'. Methods are inherited; the retrieval process involves searching through super classes of <class> if the method for <selector> is not found in <class> itself.

(10) getmethodhere

(getmethodhere <class> <selector>)

This returns the local value of the method (function name) which implements the message <selector> of <class>. If the method for <selector> is not found locally, it returns NIL.

(11) putmethod

(putmethod <class> <selector> <newvalue>)

The command *putmethod* sets the method which imple-

ments the message <selector> of the class <class> to <newvalue>. <newvalue> should be a function name.

4.2 THE RULE LANGUAGE

This section describes the language constructs provided by our system regarding the rule oriented programming.

4.2.1 FORM OF A RULE

A rule in the language describes action to be taken when specified conditions are satisfied. A rule has three major parts called the left hand side (LHS) for describing the conditions, the right hand side (RHS) for describing the actions, and the one shot description for describing the rule itself.

The syntax of a rule is :

([1] LHS RHS)

The '1' in the beginning of the rule is meta-description and is optional. If '1' is present in the beginning of a rule, then the rule will be fired only once in an invocation. If the rule is not an one-shot rule, then the following is the syntax of the rule -

(LHS RHS)

In addition, a rule can have no conditions (meaning always perform the actions) as follows :

([1] (t) RHS)

In all, a rule can be of one of the following forms :

```
(1 LHS RHS)
(  LHS RHS)
(1 (t) RHS)
(  (t) RHS)
```

LHS SYNTAX

LHS has the following format :

```
(condition1 condition2 ... conditionN)
```

The conditions in the LHS of a rule are evaluated from left to right to determine whether the LHS is satisfied. If they are all satisfied, then the rule is satisfied. For example:

```
((A B (= (+ C D) 7) (prime D)) RHS)
```

In this rule, there are four clauses on the LHS. Suppose A,B,C and D are global variables. If the values of some of the clauses are NIL during the evaluation, the remaining clauses are not evaluated. For example, if A is non-NIL but B is NIL, then the LHS is not satisfied and the third condition i.e.-(= (+ C D) 7) is not evaluated.

Each condition in the LHS should be an evaluable s-expression. All the Lisp operators are allowed in specifying the condition. The conditions should be specified in the prefix notation. For example, the condition
(= (+ C D) 7) gets satisfied only if the addition of C and D results in 7. Sending a message, calling a Lisp function or an invocation of a RuleSet can also be present in a condi-

tion.

RHS SYNTAX

The RHS of a rule consists of actions to be performed if the LHS of the rule is satisfied. These actions are evaluated in order from left to right. Actions can be invocation of RuleSets, the sending of message, Lisp function calls, variables or the 'stop' action.

RuleSets always return a value. The value returned by a RuleSet is the value of the last rule that was executed. Rules can have multiple actions on the right hand side. If the last action of a RuleSet is 'stop', then the RuleSet returns the value of the previous action. The value of a rule is the value of the last action. When a rule has no actions on its RHS, it returns NIL as its value.

4.2.2 KINDS OF VARIABLES

There are two global variables which are useful in the RuleSets. They are -

self:

It contains the current work space.

***caller:**

This contains the RuleSet name that has invoked the current RuleSet.

The language distinguishes the following kinds of variables:

RuleSet Arguments:

All RuleSets have the variable *self* as their work space. References to self can be done in the RuleSet syntax. For example, the expression (send self 'print) means to send a *print* message to *self*.

Instance and Class Variables:

All RuleSets use an object for their work space. In the LHS and RHS of a rule, the user can use or modify the instance variables and the class variables of the object contained in 'self' by the various access functions available to the user.

Temporary Variables:

Literals can also be used to refer to temporary variables allocated for a specific allocation of a RuelSet. These variables are initialized to NIL when a RuleSet is invoked. Temporary variables are declared in the **Temporary Vars** declaration in a RuleSet.

Lisp Variables:

Literals can be used to refer to Lisp variables during the invocation of a RuleSet. These variables can be global to the Lisp environment, or are bound in some calling function. Lisp variables can be used when procedure-oriented and rule-oriented programs are intermixed.

4.2.3 CREATING A RULESET

A new RuleSet can be created by sending the message *new* to the system object *RULESET*.

The syntax is -

(send 'RULESET 'new <rulesetname>)

The system interactively asks the user to enter rules and parameters corresponding to <rulesetname>.

Example.-

The following shows the session to create the RuleSet that was explained in Section 3.1 of Chapter 3.

```
<-> (send 'RULESET 'new 'checkwashingmachine)              [ret]
    (Please give the control structure-) while1            [ret]
    (Please give the While condition-)
         (not (get 'checkwashingmachine '*ruleapplied)) [ret]
    (Please give the temporary vars as a list-) ()         [ret]
    (The rule format is -)
    (<1> (cond1 cond2 ..) (action1 action2 ..))or
    (<1> (t) (action1 action2 ..))
    (End input by giving END)
    (Please give the rule-)1
    (((send self 'operational)) ((stop)))  [ret]
    (Please give the rule-)2
         ....
         ....
    (Please give the rule-)9
    (1 (t) ((stop)))   [ret]
    (Please give the rule-)10
    END  [ret]
<->
```

4.2.4 INVOKING A RULESET

One of the ways to cause RuleSets to be executed is to invoke them from rules. This is used on the LHS of rules to express predicates in terms of RuleSets, and on the RHS of

rules to express actions in terms of RuleSets. RuleSets can also be invoked from methods of objects.

The syntax of invoking a RuleSet is -

(invoke <rulesetname> <workspace>)

Example.- (invoke 'rs1 'ws1)

In this example, the RuleSet 'rs1' is invoked with the object 'ws1' as its work space. The global variable 'self' gets bound to <workspace>. So, one can refer to <workspace> through 'self' in the rules.

4.2.5 EDITING A RULESET

The following features are provided in the language.

(1) Editing the parameters (control structure, while condition, and temporary variables), rules and one-shot conditions of a RuleSet.

(2) Addition and deletion of rules from a RuleSet.

4.2.5.1 EDITING A RULE

The user can edit an existing RuleSet by sending the message *edit* to the system object *RULESET* with the RuleSet name as the argument.

The syntax is -

(send 'RULESET 'edit <rulesetname>)

Example.1-

The session given below changes the control structure of the RuleSet *checkwashingmachine* to *whileall.*

```
<-> (send 'RULESET 'edit 'checkwashingmachine)
    (Please give the corresponding no. for modifying
                                    one of the following-)
    (----- 1 parameters -------)
    (----- 2 rules ------------)
    (----- 3 oneshot conditions -------)
    1  [ret]
    (Please give the corresponding no. for modifying
                                    one of the following-)
    (---- 1 control structure ----)
    (---- 2 while condition ----)
    (---- 3 temp vars -----)
    1  [ret]
    (The present value is --) while1
    (Give the new value-) whileall [ret]
<->
```

Example.2-

The session given below is for removing the one-shot condition from the 3rd rule of the RuleSet *checkwashing-machine.*

```
<-> (send 'RULESET 'edit 'checkwashingmachine)
    (Please give the corresponding no. for modifying
                                    one of the following-)
    (----- 1 parameters -------)
    (----- 2 rules ------------)
    (----- 3 oneshot conditions -------)
    3  [ret]
    (Give the rule no. for which you want to
                toggle the one shot condition-)
    3  [ret]
<->
```

Example.3-

The session given below is for changing the LHS of the 2nd rule of the RuleSet *checkwashingmachine.*

```
<-> (send 'RULESET 'edit 'checkwashingmachine)
    (Please give the corresponding no. for modifying
                                  one of the following-)
    (----- 1 parameters -------)
    (----- 2 rules ------------)
    (----- 3 oneshot conditions -------)
    2  [ret]
    (Please give the rule no. which you like to modify)2 [ret]
    (modification 1 -lhs 2 -rhs -please give no.)1  [ret]
    ((> (getvalue self 'load) 1))
    (Pl. give the no. corresponding to the cond/act
                           you want to modify)1  [ret]
    (Please give the new version of that cond/action)
    (> (getvalue self 'load) 1.5)  [ret]
<->
```

The procedure to change an action in RHS is also very much similar to the one shown above.

4.2.5.2 ADDITION / DELETION OF A RULE

The user can add or delete a rule from the RuleSet by sending the message *add* or *del* to the system object *RULESET.* The RuleSet name should be passed as a parameter.

Example.1-

The following session shows how the user can delete the 3rd rule from the RuleSet *checkwashingmachine.*

```
<-> (send 'RULESET 'del 'checkwashingmachine) [ret]
    (Please give the ruleno. which you want to delete-)3 [ret]
<->
```

When the user adds a rule to the RuleSet, the position

at which it should be placed is also important because the rules in a RuleSet are tried from top to bottom (or from rule number 1 to rule number n). So, the system asks the user to specify the number at which the added rule should be placed. Then the earlier rules from that position would be moved down accordingly.

Example.2-

The following session shows how the user can add a new rule as the rule number 3 to the RuleSet *checkwashing-machine.*

```
<-> (send 'RULESET 'add 'checkwashingmachine) [ret]
    (What position (number) your rule should
                                    occupy?-)3 [ret]
    (Please give the rule-)
    (((not (getvalue self 'pluggedinto)))
            ((send self 'plugin)))   [ret]
<->
```

4.2.6 VIEWING A RULESET

In order to see the contents of a RuleSet (i.e. the work space class, the control structure, the while condition, temporary variable list, and the rules), the message *print* is to be sent to the system object *RULESET* along with the RuleSet name as the parameter.

Example.-

(send 'RULESET 'print <rulesetname>) [ret]

This command prints the details of the RuleSet <rulesetname> on the screen.

4.2.7 THE 'STOP' ACTION

The *stop* action can be used in the RHS of a rule to come out of the RuleSet. When this *stop* action is executed in a RuleSet, then the control returns the value of the previous action or rule (depending upon the case) and comes out of the RuleSet. The *stop* action can break out of *doall*, *while1*, and *whileall* loops also.

The syntax is:

(stop)

4.3 SAVING A SESSION

In case the user wants to save the status of the system after working on the system for some time, the user can do so by the command *loopsout*. The name of the file into which the user wants to save the system is to be given as the parameter to *loopsout*.

Example.: (loopsout 'old)

This command stores the present status of the system (including details of all the RuleSets defined, and all the objects defined) into the file names *old*. If there is already something in the file *old*, the previous contents will be overwritten by the command *loopsout*.

One can restore the status of the system by the following command.

(dskin 'old).

Chapter 5

EXAMPLES

This chapter describes three problems and shows how these problems can be solved using our system.

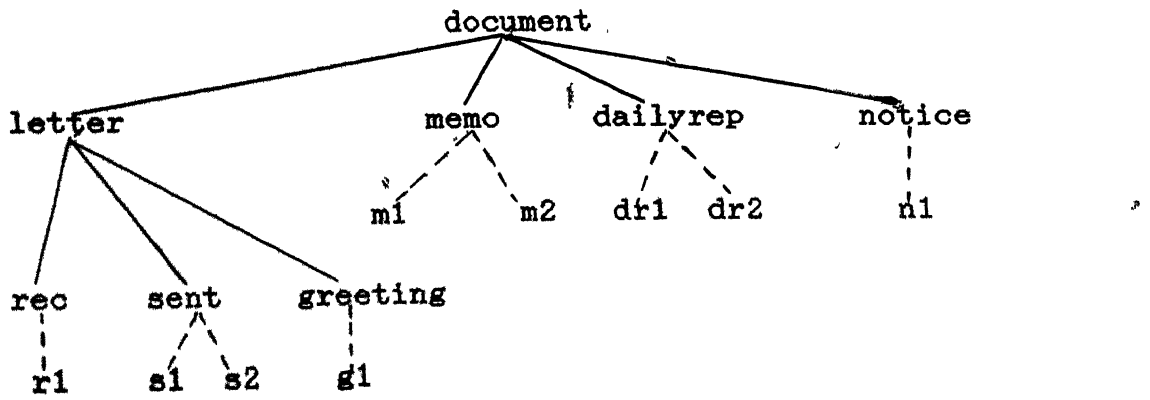
5.1 EXAMPLE-1

Problem :

In an office environment, a *document* can be a *letter, a memo, a daily-report,* or *a notice.* Letters can be of three types - received letters, letters that are to be sent out, and the received greetings cards. The problem is - what actions are to be performed when one has to deal with a particular kind of a document?.

Representation :

The hierarchical structure can be represented as follows:



document is a class with *letter*, *memo*, *dailyrep*, and *notice* as its subclasses. The subclasses of the class *letter* are *rec*, *sent* and *greeting*. -representing received letters, letters that are to be posted and the received greetings. Assume that r1 is an instance of *rec*, s1 and s2 are instances of *sent*, g1 is an instance of *greeting*, m1 and m2 are instances of *memo*, dr1 and dr2 are instances of *dailyrep*, and n1 is an instance of *notice*.

Description of the RuleSet *office* **:**

Rule-1:

If input is a document then enter a note in the Document file.

Rule-2:

If input is a letter then get the Letters file so that the letter can be stored in that file.

Rule-3:

If input is a daily-report then send it to the Director for evaluation.

Rule-4:

If input is a memo then send the memo to the concerned person and request him to give a reply immediately.

Rule-5:

If the input is a notice then display it on the notice boards.

Rule-6:

If the input is a received letter then keep it in the Letters file and if the matter is urgent, reply immediately.

Rule-7:

If the input is a received greeting then keep it in the Letters file and send Thanks reply immediately.

Rule-8:

If the input is a letter to be sent then store the xerox copy in the Letters file and post the letter immediately.

RECORDFILE

```
[1](send 'CLASS 'new 'document nil)
nil
[2](send 'CLASS 'new 'letter '(document))
nil
[3](send 'CLASS 'new 'memo '(document))
nil
[4](send 'CLASS 'new 'dailyrep '(document))
nil
[5](send 'CLASS 'new 'notice '(document))
nil
[6](send 'CLASS 'new 'rec '(letter))
nil
[7](send 'CLASS 'new 'sent '(letter))
nil
[8](send 'CLASS 'new 'greeting '(letter))
nil
```

```
[9](send 'document 'definstvars)

(Please give in the form of (variable defaultval) or
                                    <variable> one by one)
(End it by typing END)
(doc t)
END
nil

[10](send 'letter 'definstvars)

(Please give in the form of (variable defaultval) or
                                    <variable> one by one)
(End it by typing END)
(let t)
END
nil

[11](send 'memo 'definstvars)

(Please give in the form of (variable defaultval) or
                                    <variable> one by one)
(End it by typing END)
(mem t)
END
nil

[12](send 'dailyrep 'definstvars)

(Please give in the form of (variable defaultval) or
                                    <variable> one by one)
(End it by typing END)
(drep t)
END
nil

[13](send 'notice 'definstvars)

(Please give in the form of (variable defaultval) or
                                    <variable> one by one)
(End it by typing END)
(pnot t)
```

```
END
nil
[14](send 'rec 'definstvars)
(Please give in the form of (variable defaultval) or
                                    <variable> one by one)
(End it by typing END)
(rlet t)
END
nil
[15](send 'sent 'definstvars)
(Please give in the form of (variable defaultval) or
                                    <variable> one by one)
(End it by typing END)
(slet t)
END
nil
[16](send 'greeting 'definstvars)
(Please give in the form of (variable defaultval) or
                                    <variable> one by one)
(End it by typing END)
(greet t)
END
nil
[17](send 'rec 'new 'r1)
(r1)
[18](send 'rec 'new 'r2)
(r2 r1)
[19](send 'sent 'new 's1)
(s1)
[2Ø](send 'sent 'new 's2)
(s2 s1)
```

```
[21](send 'greeting 'new 'g1)
(g1)
[22](send 'memo 'new 'm1)
(m1)
[23](send 'dailyrep 'new 'dr1)
(dr1)
[24](send 'notice 'new 'n1)
(n1)
[25](send 'RULESET 'print 'office)
(The details are as follows)
(Control structure is -)whileall
(While condition is -)(not (get 'office '*ruleapplied))
(Temp variables are -)nil
(The rules are)
Rule-1
oneshot-nil
(LHS is -)
(getvalue self 'doc)
(RHS is -)
(pprint '(Enter a note in Document file))
Rule-2
oneshot-nil
(LHS is -)
(getvalue self 'let)
(RHS is -)
(terpri)
(pprint '(Get the Letters file))
Rule-3
oneshot-nil
(LHS is -)
(getvalue self 'drep)
(RHS is -)
(terpri)
```

```
(pprint '(Send it to the Director for evaluation))

Rule-4
oneshot-nil
(LHS is -)
(getvalue self 'mem)
(RHS is -)
(terpri)
(pprint '(Send the memo to the concerned person))
(terpri)
(pprint '(Request him to give a reply immediately))

Rule-5
oneshot-nil
(LHS is -)
(getvalue self 'pnot)
(RHS is -)
(terpri)
(pprint '(Put it on the notice boards))

Rule-6
oneshot-nil
(LHS is -)
(getvalue self 'rlet)
(RHS is -)
(terpri)
(pprint '(Store it in the Letters file.))
(terpri)
(pprint '(Reply immediately if urgent))

Rule-7
oneshot-nil
(LHS is -)
(getvalue self 'greet)
(RHS is -)
(terpri)
(pprint '(Store it in the Letters file))
(terpri)
(pprint '(Send THANKS immediately))
```

```
Rule-8
oneshot-nil
(LHS is -)
(getvalue self 'slet)
(RHS is -)
(terpri)
(pprint '(Take a xerox copy and store it in Letters file))
(terpri)
(pprint '(Post it immediately))

nil

[26](invoke 'office 'r1)

(Enter a note in Document file)
(Get the Letters file)
(Store it in the Letters file.)
(Reply immediately if urgent)t

[27](invoke 'office 's1)

(Enter a note in Document file)
(Get the Letters file)
(Take a xerox copy and store it in Letters file)
(Post it immediately)t

[28](invoke 'office 'g1)

(Enter a note in Document file)
(Get the Letters file)
(Store it in the Letters file)
(Send THANKS immediately)t

[29](invoke 'office 'm1)

(Enter a note in Document file)
(Send the memo to the concerned person)
(Request him to give a reply immediately)t

[3Ø](invoke 'office 'dr1)

(Enter a note in Document file)
(Send it to the Director for evaluation)t
```

CENTRAL LIBRARY
I I T. KANPUR
Acc. No. A104143

[31](invoke 'office 'n1)

(Enter a note in Document file)
(Put it on the notice boards)t

[32]

5.2 EXAMPLE-2

Problem :

Water jug A has a capacity of 5 liters, and water jug B has a capacity of 2 liters. Assume the following constraints: initially, A is filled and B is empty; the jugs are irregularly shaped, so that it is not possible to measure any intermediate amount; either jug may be poured into the other or down the drain, but no new water can be added. By what sequence of pouring, can exactly 1 liter of water be left in jug B?

Representation :

The class *sys* represents the problem domain. It has three instance variables, *a, b* and *total*. At a particular instance, the value of *a* shows how much water is present in jug A, the value of *b* shows how much water is present in jug B, the value of *total* shows how much water is present in the system in all.

Description of the RuleSet *rs1*

Rule-1:

If (a > Ø) and (b < 2) then invoke the the RuleSet 'rs2' with 'sys' as parameter.

Rule-2:

If (b = 2) then pour the water in 'b' on the floor, set 'b' to Ø, and set 'total' to (total - 2).

Description of the RuleSet *rs2*

Rule-1:

If (total >= 2) then set 'a' to (total - 2), and set 'b' to 2(ie. pour 2 liters of water from 'a' to 'b').

Rule-2:

If (total < 2) then set 'a' to Ø, and set 'b' to 'total'(ie. pour all the water of 'a' into 'b').

RECORDFILE

```
[1](send 'CLASS 'new 'sys nil)
nil
[2](send 'sys 'definstvars)
(Please give in the form of (variable defaultval) or
                                <variable> one by one)
(End it by typing END)
(a 5)
(b Ø)
(total 5)
END
nil
[3](send 'RULESET 'print 'rs1)
(The details are as follows)
(Control structure is -)whileall
(While condition is -)(neq 1 (getclassiv self 'b))
(Temp variables are -)nil
(The rules are)
Rule-1
oneshot-nil
```

```
(LHS is -)
(and (> (getclassiv self 'a) Ø) (< (getclassiv self 'b) 2))
(RHS is -)
(invoke 'rs2 self)

Rule-2
oneshot-nil
(LHS is -)
(eq (getclassiv self 'b) 2)
(RHS is -)
(print '(Pour b on floor))
(terpri)
(putclassiv self 'b Ø)
(putclassiv self 'total (- (getclassiv self 'total) 2))

nil

[4](send 'RULESET 'print 'rs2)

(The details are as follows)
(Control structure is -)do1
(Temp variables are -)nil
(The rules are)
Rule-1
oneshot-nil
(LHS is -)
(>= (getclassiv self 'total) 2)
(RHS is -)
(putclassiv self 'a (- (getclassiv self 'total) 2))
(putclassiv self 'b 2)
(print '(Pour 2 into b from a))
(terpri)

Rule-2
oneshot-nil
(LHS is -)
(< (getclassiv self 'total) 2)
(RHS is -)
(putclassiv self 'a Ø)
```

```
(putclassiv self 'b (getclassiv self 'total))
(print '(Pour all of a into b))
(terpri)

nil

[5](invoke 'rs1 'sys)

(Pour 2 into b from a)
(Pour b on floor)
(Pour 2 into b from a)
(Pour b on floor)
(Pour all of a into b)
nil

[6]
```

5.3 EXAMPLE-3

Problem :

A farmer has a wolf, a goat, and a cabbage (a very large one). He wants to get all three of them plus himself across a river, but his boat is only large enough to hold one item plus himself. How can he cross the river without leaving the wolf alone with the goat or the goat alone with the cabbage?

Representation :

The class *syst* has one instance variable *state* that shows the positions of the farmer, wolf, goat, and the cabbage. The first atom in the value indicates the position of the farmer. The second atom in the value indicates the position of the wolf. The third atom in the value indicates the position of the goat. The fourth atom in the value

indicates the position of the cabbage. *syst* has two instances *a* and *b*. The instance *a* represents the problem domain. The instance *b* is used as a dummy. *syst* has a method defined for the message *changetoopp*. The argument to this method is a subset of the set (f g w c) as a list. This method changes the position of members of the argument to the opposite side in the value of *state* of the receiver.

In this problem, a state is safe if either the farmer is on the same side as the goat or everything else is on the opposite side from the goat. The farmer can move by himself from location F to F2, leaving the other items unchanged, provided that the new location F2 is opposite F and the resulting state is safe. There are 3 other kinds of moves. The farmer takes the wolf across, the goat across, or the cabbage across. The rules should represent these points.

While deriving the steps, the changes are first made in the instance *b*, then it is checked whether the resulting state is safe and only if it is safe, the similar changes are made in the instance *a*. *w* represents the side 'west' and *e* represents the side 'east'.

Description of the RuleSet *safe* **:**

Rule-1:

If the farmer and the goat are on the same side, then it is a safe state.

Rule-2:

If the goat is on one side and the rest are on the opposite side, then the state is safe.

Rule-3:

If rules 1 and 2 fail, then the state is not safe.

Description of the RuleSet *main* **:**

Rule-1:

Copy the state of *a* into *b*. Change the position of farmer in *b* to the opposite.

Rule-2:

If the state of *b* is safe and it was not previously attained, then change the position of farmer in *a* to the opposite and add this state to the list of previously attained states.

Rule-3:

Copy the state of *a* into *b*. Change the position of farmer and wolf in *b* to the opposite.

Rule-4:

If the state of *b* is safe and it was not previously attained, then change the position of farmer and wolf in *a* to the opposite and add this state to the list of previously attained states.

Rule-5:

Copy the state of *a* into *b*. Change the position of farmer and goat in *b* to the opposite.

Rule-6:

If the state of *b* is safe and it was not previously attained, then change the position of farmer and goat in *a* to the opposite and add this state to the list of previously attained states.

Rule-7:

Copy the state of *a* into *b*. Change the position of farmer and cabbage in *b* to the opposite.

Rule-8:

If the state of *b* is safe and it was not previously attained, then change the position of farmer and cabbage in *a* to the opposite and add this state to the list of previously attained states.

RECORDFILE

```
[1](send 'CLASS 'new 'syst nil)
nil
[2](send 'syst 'definstvars)
(Please give in the form of (variable defaultval) or
                                    <variable> one by one)
(End it by typing END)
(state (w w w w))
END
nil
[3](send 'syst 'newwithvalues 'a '((state (w w w w))))
(state)
[4](send 'syst 'newwithvalues 'b '((state (w w w w))))
(state)
[5](send 'syst 'defmethod 'changetoopp '*chtoopp)
*chtoopp
```

```
[6](pp *chtoopp)

(def *chtoopp
  (lambda (l)
    (do ((y l (cdr y)) (res (getvalue self 'state)))
        ((null y) (putvalue self 'state res))
        (setq res (*chloop res (car y))))))

t

[7](pp *giveop)

(def *giveop
  (lambda (!x)
    (cond ((eq !x 'w) 'e)
          ((eq !x 'e) 'w)
          (t 'error))))

t

[8](pp *chloop)

(def *chloop
  (lambda (l x)
    (cond ((eq x 'f) (cons (*giveop (car l)) (cdr l)))
          ((eq x 'w) (cons (car l) (cons (*giveop (cadr l)) (cddr l
          ((eq x 'g)
           (cons (car l) (cons (cadr l) (cons (*giveop (caddr l))
                                              (cdddr l))
          ((eq x 'c)
           (reverse (cons (*giveop (cadddr l)) (cdr (reverse l))))
          (t 'error)))))

t

[9](send 'RULESET 'print 'safe)

(The details are as follows)
(Control structure is -)do1
(Temp variables are -)nil
(The rules are)
Rule-1
oneshot-nil
```

```
(LHS is -)
(eq (car (getvalue self 'state)) (caddr (getvalue self 'state)))
(RHS is -)
t

Rule-2
oneshot-nil
(LHS is -)
(and (eq (car (getvalue self 'state)) (cadr (getvalue self 'state))
   (eq (car (getvalue self 'state)) (cadddr (getvalue self 'state))
(not (eq (car (getvalue self 'state)) (caddr (getvalue self 'state)
(RHS is -)
t

Rule-3
oneshot-nil
(LHS is -)
t
(RHS is -)

nil

[10](send 'RULESET 'print 'main)

(The details are as follows)
(Control structure is -)whileall
(While condition is -)(not (equal '(e e e e) (getvalue self 'state
(Temp variables are -)(past)
(The rules are)
Rule-1
oneshot-nil
(LHS is -)
t
(RHS is -)
(putvalue 'b 'state (getvalue self 'state))
(send 'b 'changetoopp '(f))

Rule-2
oneshot-nil
(LHS is -)
```

```
(invoke 'safe 'b)
(not (member (getvalue 'b 'state) past))
(RHS is -)
(send self 'changetoopp '(f))
(setq past (cons (getvalue self 'state) past))
(print (getvalue self 'state))
(terpri)

Rule-3
oneshot-nil
(LHS is -)
t
(RHS is -)
(putvalue 'b 'state (getvalue self 'state))
(send 'b 'changetoopp '(f w))

Rule-4
oneshot-nil
(LHS is -)
(invoke 'safe 'b)
(not (member (getvalue 'b 'state) past))
(RHS is -)
(send self 'changetoopp '(f w))
(setq past (cons (getvalue self 'state) past))
(print (getvalue self 'state))
(terpri)

Rule-5
oneshot-nil
(LHS is -)
t
(RHS is -)
(putvalue 'b 'state (getvalue self 'state))
(send 'b 'changetoopp '(f g))

Rule-6
oneshot-nil
(LHS is -)
(invoke 'safe 'b)
```

```
(not (member (getvalue 'b 'state) past))
(RHS is -)
(send self 'changetoopp '(f g))
(setq past (cons (getvalue self 'state) past))
(print (getvalue self 'state))
(terpri)

Rule-7
oneshot-nil
(LHS is -)
t
(RHS is -)
(putvalue 'b 'state (getvalue self 'state))
(send 'b 'changetoopp '(f c))

Rule-8
oneshot-nil
(LHS is -)
(invoke 'safe 'b)
(not (member (getvalue 'b 'state) past))
(RHS is -)
(send self 'changetoopp '(f c))
(setq past (cons (getvalue self 'state) past))
(print (getvalue self 'state))
(terpri)

nil

[11](send 'a 'pinstance)

(The class is --)syst
(The instance vars -- The values)
state(---------)(w w w w)
t

[12](invoke 'main 'a)

(e w e w)
(w w e w)
(e e e w)
(w e w w)
```

```
(e e w e)
(w e w e)
(e e e e)
nil
```

[14]

Chapter 6

THE IMPLEMENTATION

The system is implemented in the C-Lisp environment available on top of Franz-Lisp on HCL-Hórizon III. This chapter gives details of the implementation of the language that was described in the Chapter 4.

6.1 DATA STRUCTURES

There are four kinds of objects in the system. They are:

(1) Metaclasses.

(2) Classes.

(3) Instances.

(4) RuleSets.

6.1.1 METACLASSES

Each metaclass <m-class> in the system has the following properties defined on it.

**is-mclass*

It is 't' for all metaclasses.

**lclass*

This contains the list of classes defined under the

metaclass <m-class>.

**methods*

This contains the list of messages that the metaclass <m-class> can receive i.e,there exist methods corresponding to the selectors present in this list for the metaclass <m-class>.

<selector>

This property contains the name of the function that is to be applied when the metaclass receives the message <selector>. While defining the method using *defmethod* for the message <selector>, if the user chooses to give the 'form' of the method, then the system creates a function with the name *<m-class>~<selector>* and puts the form in that function. This name is stored under the property <selector> instead of the form.

6.1.2 CLASSES

Every class <class> in the system has the following properties defined on it.

**is-class*

It is 't' for all classes.

**clvars*

This contains the list of class variables that are defined in the class <class>.

**invars*

This contains the list of instance variables that are defined in the class <class>.

**mclass*

This contains the name of the metaclass of the class <class>. In general, the system object 'CLASS' is the metaclass for most of the classes.

**methods*

This contains the list of messages the class <class> can receive.

**insts*

This contains the list of names of the instances that are defined under the class.

**supers*

This contains the list of names of the super classes of the class <class>.

**subs*

This contains the list of names of the sub-classes of the class <class>.

<class-var-name>

This property contains the value of the class variable <class-var-name> in the class <class>.

<inst-var-name>

This property contains the default value of the instance variable <inst-var-name> in the class <class>.

This property contains the local value of the instance variable <inst-var>.

6.1.4 RULESETS

Each RuleSet <rs> has the following properties defined on it.

**is-ruleset*

It is 't' for all RuleSets.

**control*

This contains the value of the parameter *control structure* of the RuleSet <rs>.

**while*

If the control structure of <rs> is either 'while1' or 'whileall', this property contains the 'while condition' for the RuleSet <rs>. Otherwise, it contains NIL.

**tempvars*

This property contains the list of temporary variables that can be used in the RuleSet <rs>.

**norules*

This property contains the number of rules present in the RuleSet <rs>. This is useful in implementing the *add* and *del* methods of RULESET because these methods require to move up or move down certain number of rules in the RuleSet.

ruleapplied

It becomes 't' when at least a rule gets fired in the RuleSet <rs> in an invocation.

rulesfired

It contains the list of numbers of the rules that were already fired in a particular invocation. So, whenever a RuleSet is invoked, this property holds the value NIL initially.

The rules are stored as properties under the atom *<rs>~<number>*. The 'nth' rule in the RuleSet <rs> is stored under the atom *<rs>~n*. Each rule *<rs>~n* has the following properties.

lhs

This contains the LHS part of the rule.

rhs

This contains the RHS part of the rule.

oneshot

It is 't' if the rule is a 'one-shot' rule.

6.1.5 GLOBAL VARIABLES

The following are the list of global variables used in the system.

self

This contains the name of the object that is to be worked upon.

*class

This contains the list of classes defined under the metaclass 'CLASS'. This variable is useful in disking out a session.

*metaclass

This contains the list of metaclasses defined in the system.

*rulesets

This contains the list of RuleSets defined in the system.

*caller

In a RuleSet <rs>, if 'send' or 'invoke' is executed as an action, then this global variable is set to the RuleSet <rs>. This is useful in implementing the 'invoke' function.

*return

This contains the value returned by the last action in an invocation of a RuleSet.

*stop

This is set to 't' when the action 'stop' is executed in an invocation of a RuleSet. The action 'stop' returns the value of the global variable '*return'.

*fromsend

Its value is 't' as long as the control is in the

'send' function. This is useful in implementing the 'invoke' function.

**estack*

This is a very important global variable. This can be called as the 'system stack' of our system. It looks as shown below.

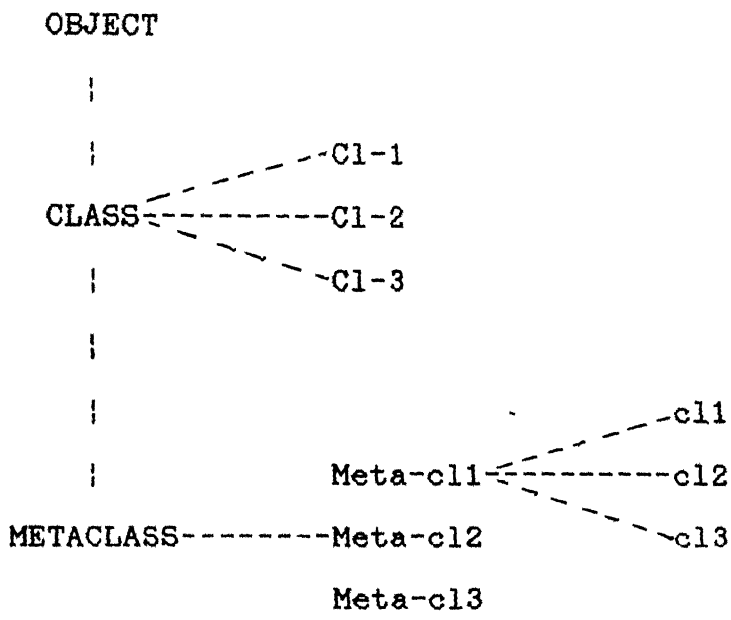
(<element-1> <element-2> ... <element-n>)

Each element can be a name of an object (self), or a list containing the status of a RuleSet that is stored when the RuleSet executes a 'send' or an 'invoke' action. The status of the RuleSet contains the information regarding work space object, RuleSet name, temporary variables as an associated list of variables and values, list of numbers of the rules that were fired in the invocation till that instance and the value of '*ruleapplied' property of the RuleSet.

6.2 OBJECT, CLASS, METACLASS AND RULESET

OBJECT, CLASS, METACLASS and RULESET are system objects. They have standard methods defined on them.

HIERARCHY



OBJECT is super of CLASS. CLASS is a metaclass. The instances of CLASS are classes. CLASS is super of METACLASS. METACLASS is a meta-meta-class, meaning it's instances are metaclasses. The instances of a metaclass are classes. The only use of metaclasses is that the user can redefine some of the standard methods of the system by placing them as methods of the metaclass.

Example-

CLASS has a standard method for the message 'new'. The user may want to add some additional actions <action1> <action2>...<action-n> to what is done by the standard method of CLASS. Then the user can define a method for 'new' in the metaclass <meta-cl>. A sample session is shown below.

```
<1> (de <meta-cl>~new (obj supers)
      (send 'CLASS 'new obj supers)
```

```
        <action1>
        <action2>
           .
           .
        <action-n>
      ) [ret]
<2> (send <meta-cl> 'defmethod 'new '<meta-cl>~new) [ret]
<3> (send <meta-cl> 'new <cl-name> <super-list>) [ret]
```

Now, the command number 3 creates a class with name <cl-name> under <meta-cl> according to the definition of '<meta-cl>~new'.

While inheriting the methods for a particular message, metaclasses are tried before the object CLASS.

Example-

Suppose, supers of a class A are M and N. The metaclass of A is 'MA'. Suppose, at the beginning of a particular iteration while inheriting the methods through the depth-first algorithm, the *class-list* that contains the names of classes to be tried is (A B C). Suppose, the class M has no supers and its metaclass is 'MM'. Then the values of *class-list* in subsequent iterations are given below :

```
(A B C)
(M N B C MA)
(N B C MA MM)
   .
   .
```

Only when *class-list* becomes NIL (i.e, all the metaclasses are already tried), the object CLASS is tried.

OBJECT has methods to handle instances. CLASS has

methods to handle classes. METACLASS has a method to create metaclasses. RULESET has methods to handle RuleSets. Access functions are not defined as standard methods.

6.2.1 METHODS OF OBJECT

The following methods are defined in the system object OBJECT.

new

It creates a new instance under a class.

newwithvalues

It creates a new instance with some of its variables initialised, under a class.

listinst

This method prints the list of names of instances of a class.

classlist

It gives the list of names of classes defined under a metaclass.

6.2.2 METHODS OF METACLASS

A method for the message 'new' is defined in the object METACLASS.

new

This method creates a metaclass under METACLASS.

6.2.3 METHODS OF CLASS

The following methods are defined in the system object CLASS.

new

This method creates a class under CLASS or a metaclass.

definstvars

It takes the details of instance variables of a class from the user.

defclvars

It takes the details of class variables of a class from the user.

defmethod

It is for defining a method for a class.

rmmethod

It is for removing a method from a class.

addsuper

It adds the specified supers to the supers property of a class.

rmsuper

It removes the specified supers from the supers property of a class.

rmsub

It removes the specified subclasses from the subs property of a class.

pclass

It prints the details of a class.

pinstance

It prints the details of an instance.

6.2.4 METHODS OF RULESET

The following methods are defined in the system object RULESET.

new

This creates a RuleSet.

edit

This method is for editing a RuleSet.

del

This is for deleting a rule from a RuleSet.

add

This is for adding a rule to a RuleSet.

print

This prints the details of a RuleSet.

6.3 SEND - INVOKE ALGORITHM

The function 'send' is for sending a message and the function 'invoke' is for invoking a RuleSet. Between these two functions, four cases of calling functions are possible. They are-

(1) 'send' in 'send'.
(2) 'send' in 'invoke'.

(3) 'invoke' in 'send'.

(4) 'invoke' in 'invoke'.

The algorithms should take care of all these possibilities. The algorithms for 'send' and 'invoke' are given below.

send Algorithm-

```
(1) *fromsend = t.
(2) If *caller then
    {
       ;send in invoke case
       p = 1.
     - Push the <stack-element> of *caller on *estack.
       *caller = ().
    }
(3) Push the object name (i.e, self) to which the message is
    sent on *estack.
(4) Apply the method (inherit if not defined in 'self').
(5) Delete the top element from *estack.
(6) If (p = 1) then
    {
      ;send in invoke case
      -Restore the status of the RuleSet using *estack.
      -Delete the top element from *estack.
    }
(7) *fromsend = NIL.
```

invoke Algorithm-

```
(1) If *caller then
     ;invoke in invoke case.
     p = *caller.
     - Push the <stack-element> of *caller on *estack.
(2) *caller = (Present RuleSet).
    self = (Object that is passed as the parameter).
(3) Initialize all the parameters of the present RuleSet.
(4) If *fromsend then
```

```
      ;invoke in send case.
      q = t;
(5) Apply the control structure.
        (do1, doall, while1 or whileall)
(6) If q then
      self = (The first element of *estack).
      *caller = ().
(7) If p then
     ;invoke in invoke case.
      -Restore the status of the RuleSet using *estack.
      -Delete the top element from *estack.
(8) If (not p) and (not q)
     ;invoke is called from top-level.
     *caller = ().
```

6.4 LOOPSOUT ALGORITHM

The algorithm given below describes how the function 'loopsout' is implemented in the system.

Algorithm-

```
(1) Store the values of the global variables -
          - *class
          - *metaclass
          - *rulesets
(2) For each metaclass <m-class> do
     {
       - Store the plist of <m-class>.
       - Store the functions corresponding to the methods
         of <m-class>.
     }
(3) For each class <class> do
     {
       - Store the plist of <class>.
       - Store the plist of each instance of the class <class
       - Store the functions corresponding to the class
```

```
        <class>.
    }
(4) For each RuleSet <rs> do
    {
      - Store the plist of the RuleSet <rs>.
      - For each rule '<rs>~<n>' do
          {
            - Store the plist of the rule <rs>~<n>.
          }
    }
```

6.5 HOW TO START THE SYSTEM

The following sequence of commands are to be given to work on the system.

```
% ~sangal/clisp.exe
<-> dskin loops [ret]
<->
```

If the previous session was stored into a file, the user has to 'dskin' that particular file to restore the status of the system.

6.6 A NOTE

In the code of the system, all the functions and the system properties have their names starting with a '*'. All the variables have '!' as the first character in their names. So, the user should try to avoid defining Lisp global variable names starting with a '!' and function names starting with a '*'.

The names of classes, instances, metaclasses, and RuleSets should not clash because properties are defined on

the names. One way of avoiding the name clashing is to have all names of classes starting with a 'c', all names of instances starting with an 'i', all names of metaclasses starting with an 'm', and all names of RuleSets starting with an 'r'.

Chapter 7

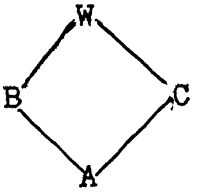
CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

We have presented a system for object and rule oriented programming in this report. The basic concepts, the constructs and the implementation details are discussed. We have given examples to show how a user can get about solving problems using our system. The system is implemented in C-Lisp on HCL-HorizonIII.

There is a scope for future work in this topic. The following are the suggestions for further work:

(1) In our system, the inheritance mechanism is implemented through a depth-first search strategy. In some cases, this may lead to redundant checks.

Example: Suppose, the class W is super of the classes B and C, and B and C are supers of the class A.



Suppose, the *class-list* consists of A only, initially. Then, the values of *class-list* after each iteration are given below:

(A)

(B C)
(W C)
(C)
(W)
()

We can see that the class W is checked twice. So, one can try to come up with new algorithms that take care of this redundancy.

(2) In our system, there is no inheritance provided amongst RuleSets. One can think of providing inheritance in RuleSets also, so that large RuleSets can be constructed according to an object-kind of hierarchy.

(3) The control structures *doI*, *doall*, *whileI*, and *whileall* are available for RuleSets in our system. One can develop some new control structures depending upon the nature of the problem. The code of our system is written in a manner that enables easy addition of new control structures.

(4) The RuleSets of our system try to fire rules from top to bottom (i.e. Rule number 1 to Rule number N) in an invocation. Instead, certainty factors can be attached to the rules of a RuleSet by the user. Then, the shell evaluates the certainty factors of all the rules, ranks them and fires the rules in decreasing order of certainty factor.

(5) In our system, while invoking a RuleSet, only one object can be given as an argument to the RuleSet.

Instead, one can provide the user with the facility of giving more than one argument while invoking a RuleSet.

REFERENCES

[1] Adele Goldberg, David Robson.

Smalltalk-80 : The language and its Implementation. *Reading, Addison-Wesley.1983.*

[2] Adrian Walker, Micheal Mc Cord, J.F.Sowa,W.G.Wilson.

Knowledge Systems and Prolog. *Reading, Addison-Wesley.1987.*

[3] Daniel G.Bobrow, Mark Stefik.

The LOOPS Manual. *Xerox Corporation.1983.*

[4] Forgy,C.L.

OPS5 User's Manual. *Technical Report CMU-CS-81-135, Carnegie-Mellon University.1981.*

[5] Shinji Yokoi.

A Prolog Based Object Oriented Language SPOOL and its Compiler. *LNCS - 264, Logic Programming '86.*

[6] Timothy Budd.

A Little Smalltalk. *Reading, Addison-Wesley.1987.*

SUMMARY OF COMMANDS

(I) The Object-Oriented Programming Constructs

(1) Send :

(send <obj> <sel> <arg1> ... <argn>)

The message <sel> is passed to <obj> with the arguments as parameters.

System defined selectors:

new

To create a new object.

newwithvalues

To initialize instance variables while creating an instance itself.

definstvars

To define instance variables for a class

defclvars

To define class variables for a class.

defmethod

To define a method for a class.

rmmethod

To remove a method from a class.

addsuper

To add supers to a class.

rmsuper

To remove supers from a class.

rmsub

To remove subclasses from a class.

pclass

To see the details of a class.

pinstance

To see the details of an instance.

listinst

To get all the names of instances defined under a class.

classlist

To get all the names of classes defined under a meta-class.

destroy

To remove an object from the environment.

(2) The Access Functions

getvalue

To get the value of an instance variable from an instance.

getclassvalue

To get the value of a class variable from a class.

putvalue

To put a new value in an instance variable of an instance.

putclassvalue

To put a new value in a class variable of an class.

getclassiv

To get the default value of the instance variable of an object.

putclassiv

To put a new value as the default value of an instance variable of an object.

getivhere

To get the local value of an instance variable of an object.

getcvhere

To get the local value of a class variable of an object.

getclass

To get the name of the class to which the object belongs.

getmetaclass

To get the name of the metaclass to which the object belongs.

getmethod

To get the method (Lisp function name) which implements a message of a class.

getmethodhere

To get the local value of the method which implements a message of a class.

putmethod

To put a new value (a function name) as the method for a message in a class.

(II) The Rule Language

(1) (send 'RULESET <sel> <rulesetname>)

The selector <sel> is passed to the system object RULESET with <rulesetname> as the argument.

System defined selectors

new

To create a new RuleSet.

edit

To edit a RuleSet.

add

To add a rule to a RuleSet.

del

To delete a rule from a RuleSet.

print

To see the details of a RuleSet.

(2) (invoke <rsname> <workspace>)

This invokes the RuleSet <rsname> with <workspace> as the current work space.

(3) (stop)

This makes the control come out of the execution of a RuleSet. This command can break out of do1, while1 and whileall.

(III) (loopsout <filename>)

This command saves the session on to the file <filename>.